Hirun Hettigoda Fiery Dragon Design

Video link:

https://www.youtube.com/watch?v=Dx0ACUj09A0

Instructions for creating and running jar file:

- the application requires JDK 22 and the latest version of JAVA downloaded from ORACLE
- to build executable within intellij go on file, project structure, artifact, +, JAR, choose the main file, apply and ok
- to actually build it: go to build, build artifact, the jar file created represented above and build
- the application should now run as an executable

References:

- used ChatGPT to create the images represented within the game

(i) Set up initial game board (including randomised positioning for chit cards):

a. In the UML class diagram, it is illustrated that the Game class creates the actual board used. In the Board class there are three different elements consisting of an array of volcano cards, caves, chit cards, and players. When initially picturing the board, I put only the Volcano cards since that is essentially the board, but to my understanding caves, chit cards and players are also relevant under creating the board. In the Board class it takes the volcano cards, chit cards and caves elements into account to join everything together in the desired format to create the board. There are separate classes for Cave, DragonCard and VolcanoCard since there are methods in these classes along with attributes used in the Board class. The randomisation of the dragon cards are represented through a method in the DragonClass called shuffleDragon() which uses the in built Collections class to shuffle it. This method is then called in the Board class. Finally, the addPlayers method under the Board class is used to add the number of players in the game to certain caves on the board, which varies depending on the number of players being chosen. This method will be called in the Main/Game class. The board creation is done through JPanels, JLabels and JButtons with a specific layout manager like GridBagLayout or BorderLayout being followed.

(ii) Flipping of dragon (chit) cards:

a. Flipping the chit card is a method under the DragonCard class. I've broken the flipping essence in which the chit card can be flipped up in which it represents the creature and the creature quantity. Under the DragonCard class there is also a flipStatus attribute, when a chit card is flipped up the flip status would be set to True, whilst when it is down it is set as False. Flipping a chit card is something the player would do when playing their turn, this is represented through the dependency relationship with the Player class and the DragonCard class. When the dragon card is facing down there is a general image on it, however when it is flipped an image of what's under the DragonCard would be represented. The action is represented through the addActionListener method as well as revalidate() and repaint().

(iii) Movement of dragon token based on their current position as well as last flipped chit card:

a. The movement of the dragon token is based on a few different elements with the movePlayer method being the main method when the playTurn method is running. The movePlayer method is used to move the player either backwards, forwards or no movement. The movePlayer method considers the chitMatch method. This method checks whether the creature on the chit card flipped, aligns with the creature of the current position of the player. If there is a match the player moves forward the quantity of the creature which is represented on the dragon card and the currentPosition is incremented accordingly. If there isn't a match the movePlayer method doesn't move the player anywhere. Finally, if the chitCard flipped by the player is a pirate creature the player is moved back regardless of their current position creature with their currentPosition decremented accordingly unless they are in a cave.

(iv) Change of turn to the next player

a. This functionality is represented through the switchPlayer method under the Player class. Once the current player is having their turn which is represented under the playTurn method, the switchPlayer method will take place if the chitMatch method returns False. This means the current player taking their turn flipped over a chit card which doesn't correspond to the current volcano card tile creature they are on. This then sets the turnStatus of the current player to False and sets the turnStatus of the next player to True. In the future there may be time components involved which sets a certain amount of time a player has during their turn. When the timer is done the switchPlayer method would again apply in this case.

(v) Winning the game:

a. It is possible for only a single player to win the game at this point. Winning the game is done when a player goes from their cave around the whole board and back to their cave. In my Player class I have attributes representing the startPosition, currentPosition and endPosition as integers. Once the player has traversed through all volcano cards with their currentPosition being equal to the endPosition the player has won. This will allow the win method to deliver a message/prompt that a certain player has won, which would then restart the game. In the future there may be features that allow second place, third place etc which would allow only a prompt to pop up without the game having to restart when a single player wins.

Design Rationales:

- Two classes I have included are the DragonCard class and the DragonToken class. I created a DragonCard class since I thought it would involve many unique methods inside it, which it did. Like flipping of a DragonCard or randomising of a DragonCard. In my opinion when designing the game, it wouldn't make sense to make it a method or disregard it as a class since it is an important factor of the game. The DragonToken class was a unique method due to significance it has in the game as well as consisting of several methods. In my opinion disregarding a class like DragonToken would cause issues in the future and it would not make sense to have it as a method since it is a unique entity. There are a few methods in it like switchPlayer or chitMatch which may be

relevant in another class when implementing further, however for now I see it as a method in this class.

- One key relationship in my class diagram includes the BoardPanel class and the DragonCard class. The DragonCard being a key component in the game as a whole and within the Board class it allows the game to function. Factors like flipping a dragon card and simply displaying it on the map allows for the game to flow in the best way in my opinion. The relationship between these two classes is what makes the game feel alive in my opinion. Another key relationship is between the Game/Main class and the BoardPanel class. In my opinion there weren't much aggregation or composition relationships. It can be argued that the direct relationships with the board and components like volcano cards could be however they don't solely rely on those classes. This could be the same for the Game and BoardPanel class however for now I see it as a direct association since an instance of the board is called in the Game class.
- In my design I did decide to use inheritance. I created an abstract Creature class in which different types of creatures like Spider can branch off it. In the beginning I thought an enum was the right choice however it may prevent extra functionality in the future resulting in me to use inheritance. For the time being I also made a SpecialCreature abstract class. This is also since there may be other new special creatures which could be added with different abilities. In my opinion they are also a bit different compared to the normal creature, which is why I didn't just branch off from there.
- One cardiniality within the game is between the BoardPanel class and the DragonCard class representing 1 and 16. This represents that 1 BoardPanel would contain exactly 16 DragonCards. It wouldn't be 1..16 or 1..* since the game has a fixed amount of dragon cards at this point in time where there cannot be more than 16 or less than 16. The same logic applies for the other components within the BoardPanel.

Design Patterns:

Design patterns that is visible in the design may include the singleton pattern in which classes like Game or BoardPanel are intended to have only one instance throughout the applications cycle. Although not explicitly demonstrated at the moment there could be use of the factory pattern design due to the classes like DragonCard and Creature in which it may become more complex down the track. Therefore, having a design pattern of this sort could help encapsulate the instantiation logic. A pattern I haven't used in the game is the strategy pattern since in this game certain behviours can be handled through subclassing and overriding methods without having code duplication so the strategy pattern may introduce unnecessary abstraction.